Information and Software Technology xxx (2011) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Controlled composition and abstraction for bottom-up integration and verification of abstract components

Yunja Choi [a,*], Moonzoo Kim [b]

[a] School of Computer Science and Engineering, Kyungpook National University, Daegu, Republic of Korea
[b] Department of Computer Science, KAIST, Daejeon, Republic of Korea

ARTICLE INFO

Article history:
Received 6 July 2010
Received in revised form 15 July 2011
Accepted 15 August 2011
Available online xxxx

Keywords:
Controlled composition
Abstraction
Verification

ABSTRACT

This work proposes a method for improving the scalability of model-checking compositions in the bottom-up construction of abstract components. The approach uses model checking in the model construction process for testing the composite behaviors of components, including process deadlock and inconsistency in inter-component call sequences. Assuming a single processor model, the scalability issue is addressed by introducing operational models for synchronous/asynchronous inter-component message passing, which are designed to reduce spurious behaviors caused by typical parallel compositions. Together with two abstraction techniques, synchronized abstraction and projection abstraction, that hide verified internal communication behavior, this operational model helps to reduce the complexity of composition and verification.

The approach is supported by the Marmot development framework, where the soundness of the approach is assured through horizontal verification as well as vertical verification. Application of the approach on a wireless sensor network application shows promising performance improvement with linear growth in memory usage for the vertically incremental verification of abstract components.

1. Introduction

Model checking [17] has been actively used to identify hard-to-identify problems, such as process deadlock, data race, and inconsistency between inter-component call sequences, using exhaustive search techniques over the system state space. However, it suffers from the notorious state-space explosion problem that the memory and time requirements for the verification increase exponentially as the size of the search space increases. Though recent technical advances have shown gradual improvements regarding the scalability issue [15,2,8,42], model checking is still considered impractical to routinely apply to real-life software applications.

As scalability improvement becomes a key factor for the applicability of model checking in practice, integrating model-checking techniques into the component architecture has been an active research area, following the divide-and-conquer principle [9,30,42,18,49,48]. Nevertheless, most of the existing approaches do not address the scalability issue together with the communication overhead caused by compositions. We tackle this issue using a systematic behavioral composition of components on a single processor model.

This work proposes a systematic method for alleviating the scalability issue in two ways. First, an operational model for the behavioral composition of abstract components is defined. The operational model classifies the message passing mechanism into two types, synchronous message passing and asynchronous message passing. Synchronous message passing constrains the interaction behavior in such a way that a message sender waits for the return of its result from the message receiver, deactivating the currently active configuration temporarily. The operational model reduces spurious composition behaviors compared to those with typical parallel composition, where processes are allowed to randomly interleave with each other, but it does not change the original functional behavior if the system works on a single processor – a typical case for safety–critical embedded software where multi-threading is not common. We call such a composition with the operational model a *controlled* composition. Second, greater reduction is achieved by abstracting verified component behavior using *synchronized abstraction*, which leaves only the functional behavior of the composite component, and *projection abstraction*, which removes internal behavior that is not provided by the external service structure after composition.

Each abstract component resulting from the controlled composition and abstraction constitutes a representative of the set of composed components. It is also considered as a unit of next-level composition in the successive model construction process. We

* Corresponding author.
E-mail addresses: yuchoi76@knu.ac.kr (Y. Choi), moonzoo@cs.kaist.ac.kr (M. Kim).

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.08.001

have formulated these abstraction techniques with respect to the composition of abstract components in the process of bottom-up composition of behavioral models. Though a number of techniques are available for extracting low-level behavioral models from code [5,26,37], our approach is new in that it provides systematic bottom-up model extraction and verification up to the top-level abstract components.

This approach was developed on top of the existing Marmot framework for component-based development of embedded software [14]. The framework is equipped with supporting tools, including automated environment generation for a verification unit and translation into the back-end model checker Spin. We added automation for controlled composition in this framework, but abstraction is done manually.

We note that this work neither aims at code verification nor at providing a methodology for component-based development. If appropriate, existing code verification tools [6,16,47,20] may be used instead of Spin in our verification framework. We assume that information on unit components and their composition structure is already available prior to applying our approach. Our ultimate goal is to provide a systematic method for reverse-engineering models from code to support high-level reuse in the existing component-based development process. This work can be considered as a preliminary step towards achieving this goal.

The main contribution of the work can be summarized as follows:

1. A systematic compositional minimization method specifically designed for single processor systems is introduced and the effectiveness of the approach is demonstrated with a series of experiments that showed linear growth of memory usage instead of typical exponential growth.
2. An iterative behavioral model extraction technique is provided, which enables bottom-up composition and localized verification from physical unit components to the top-level abstract component. A novel application of the technique on a TinyOS application is demonstrated.

The novelty of the approach lies in the fact that the improvement in model checking scalability becomes more pronounced as the bottom-up behavioral composition is successively abstracted. Once the verification of the controlled composition is done, the detailed realization of each abstract component is hidden, leaving only its specification behavior. In this way, the verification of a higher-level abstract component involves only its immediate sub-components, localizing the verification problem.

The effectiveness of the suggested approach is demonstrated with a series of experiments on a wireless sensor network application that is available with the distribution of TinyOS [1], an open-source operating system for wireless sensor networks. The experimental results show that the performance of the model checking shows linear growth in memory usage as the level of abstraction grows vertically – a promising improvement over the typical exponential growth. To the best of our knowledge, this is the first case of systematically applying bottom-up composition and verification from physical operating system components to an application component.

The remainder of this paper is organized as follows: Section 2 provides an overview of our approach. Section 3 defines our notions of component. Sections 4 and 5 propose operational models for controlled composition and abstraction techniques, respectively. Section 6 describes our verification framework, followed by the comparative experimental result on model checking a TinyOS application using the suggested approaches (Section 7). We conclude with a summary on related work in Section 8 and a discussion in Section 9.

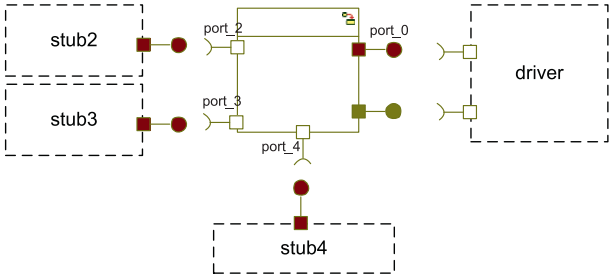


Fig. 1. Environments for unit verification.

2. Overview of the proposed approach

Component-based verification approaches help to localize problems and reduce verification complexity using the divide-and-conquer strategy. However, typical software may consist of dozens to hundreds of components, which results in complex and intractable interaction behavior.

A similar difficulty may occur when model checking a unit component, since we also have to consider its environment. As illustrated in Fig. 1, the verification of a unit component may require creating stubs and test drivers; in the figure, the component supports two *provide* ports, which are supposed to be used by its drivers, and three *use* ports, which specify the required services provided by external components. The stubs, acting as components that process messages from the unit component, and the drivers, acting as stimuli generators, connected to these ports may have their own independent behavior. Regardless of how simple they may be, composition of these independent processes may result in complex interleavings, and, thus, lead to state-space explosion.

This work deals with this issue in three ways: First, the behavior of a composite component is controlled by defining operational models for synchronous/asynchronous inter-component message passing mechanisms (see Section 4). Second, we reuse verified components in higher-level compositions after abstracting internal communication behavior and uninteresting behavior for the new composition (see Section 5). Third, this verification-abstraction-reuse process is performed iteratively, which supports a systematic verification cycle (see Section 6).

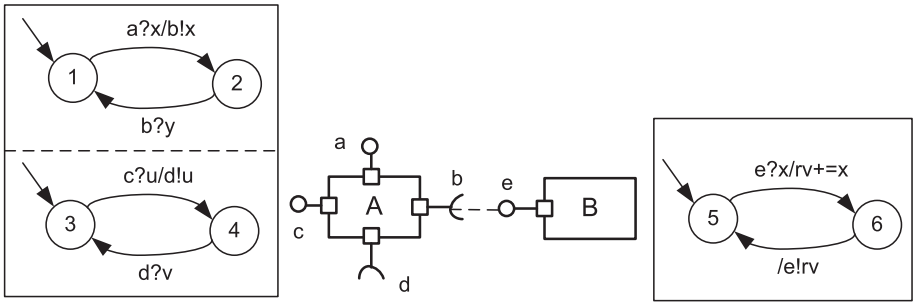
2.1. Controlled composition

We introduce operational models for compositions of communicating components with the examples in Figs. 2 and 3. In Fig. 2a, the structural composition of the two components A and B is depicted in the center, while the behavioral model of each component is specified in statecharts [31] on the left/right side. The two components A and B are composed by connecting port b of A and port e of B. In the statecharts, a transition is specified with events and actions, where $a?x$ represents a message arrival event at port a and $b!x$ represents a message sending action at port b. We represent a transition from a source state s to a target state t with a triggering event p and an action q as $s \xrightarrow{p/q} t$.

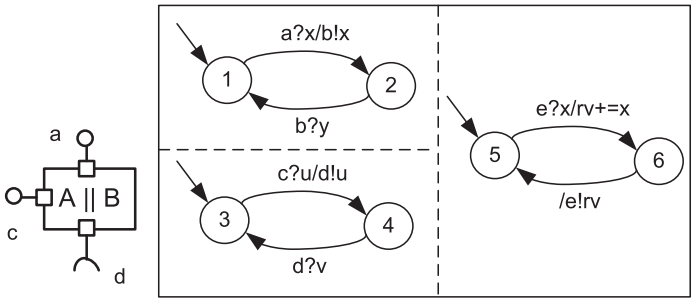
If we apply typical parallel composition, the resulting composition behavior would look like the statechart in Fig. 2b. Note that any combination of the three independent sets of transitions is possible. For example, the sequence of transitions $(1,3,5) \xrightarrow{a?x/b!x} (2,3,5) \xrightarrow{c?u/d!u} (2,4,5) \rightarrow \ldots$ is possible, since the active configuration of the composite component after the first transition is {2,3,5}, meaning that any outgoing transitions from these three states are possible for the next transition.[1] However, this sequence

[1] For notational convenience, we represent a set of parallel transitions in one transition; e.g., $(1,3,5) \xrightarrow{a?x/b!x} (2,3,5)$ means $(1) \xrightarrow{a?x/b!x} (2) \wedge (3) \xrightarrow{a?x/b!x} (3) \wedge (5) \xrightarrow{a?x/b!x} (5)$.

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

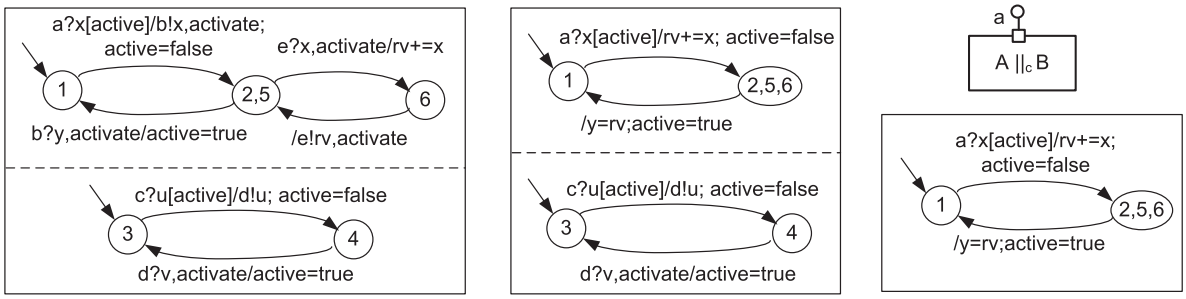


(a) Typical structure and behavior of two components



(b) Parallel composition

Fig. 2. Typical composition.



(a) Controlled composition (b) Synchronized abstraction (c) Projection abstraction

Fig. 3. Controlled composition.

of transitions is not possible on a single processor if $b!x$ (which is to be bound with $e?x$) is a synchronous call, i.e., if component A sends out message x through port b and waits for the return of the control. This case limits the possible sequence of transitions; after the transition $(1,3,5) \overset{a?x/b!x}{\rightarrow} (2,3,5)$, the only possible transition should be $(2,3,5) \overset{e?x/rv+=x}{\rightarrow} (2,3,6) \overset{/e!rv}{\rightarrow} (1,3,5)$. In other words, the active configuration after the first transition is {2,5}, since synchronous message sending prevents state 3 from being active till it receives a return message.

The behavior anticipating synchronous message passing can be modeled as in Fig. 3: Initial states of a component can be either in an active state or in a passive state, depending on whether the component itself is active or not. A component in an active state can send out a synchronous message call together with the activation message *activate*, and then deactivates itself by setting the *active* flag false. This forces the component to be in a passive state until the return message comes with an activation message from the same port used to send the message. If we impose such a control sequence, the transition from state 3 to state 4 is not possible when the state (2,5) is the current state because the component is not in an active state.

We combine this operational model with the verification model depicted in Fig. 1. Initially, the only active component is the driver, which generates external stimuli infinitely. However, we assume that the unit component handles an external stimulus infinitely faster, adopting the synchrony hypothesis; in other words, the computational speed of each component is faster than the speed of stimuli generation by the external environment. The stubs and unit components are passive components that are ready to receive messages together with the activation signal. Note that the operational model serializes the flow of control for synchronous message passing, removing spurious behavior. This is valid for a single processor system, which is the typical case for most safety–critical embedded software; typical parallel composition is an over-approximation in such systems.

2.2. Abstraction

The controlled composition removes spurious behaviors but retains the actual behaviors of the composite components, including internal communications. After verification of the controlled composition, further abstraction can be performed focusing on

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

the functional behavior; *synchronized abstraction* removes internal communication behaviors, replacing them with local value assignments. *Projection abstraction* hides uninteresting internal behavior of a composite component with respect to its externally provided services.

Fig. 3b illustrates the synchronized abstraction of the controlled composition in Fig. 3a. Based on the structural information that ports b and e are connected, we can combine the sending action $b!x$, *activate*; *active = false* in the transition $(1) \rightarrow (2,5)$ and the triggering event $e?x$, *activate* in the transition $(2,5) \rightarrow (6)$ into one. A similar reduction is possible for the sending action $e!rv$, *activate* in the transition $(6) \rightarrow (2,5)$ and the triggering event $b?y$, *activate* in the transition $(2,5) \rightarrow (1)$.

Fig. 3c shows *projection abstraction* applied after the synchronized abstraction; assuming that the composite component $A\|_cB$ needs to provide the services specified in port a only (and the system is not interested in the services provided by port c), the internal behavior related to port c, the transition $(3) \rightarrow (4)$, can be removed. The state 4 and its outgoing transitions are also removed, since there are no more incoming transitions to the state, and, thus, it is not reachable from the initial state.

2.3. Verification

We first verify unit components composed with their drivers and stubs. Controlled composition of unit components replaces the stubs with unit components incrementally, where the composition is determined by the composition structure and the type of inter-component communication. A set of components may be composed to build an abstract component using controlled composition, producing the realization behavior of the abstract component. Synchronized abstraction and projection abstraction are applied after the controlled composition is verified, producing the specification behavior of the abstract component. Synchronized abstraction is an effective way to reduce the complexity of the behavior model, but it removes communication details. Therefore, this technique is applied only after the communication aspect has been verified in controlled composition.

Fig. 4 illustrates a typical composition structure of a system where the bottom-level boxes represent physical components and the upper-level boxes represent abstract components composed of lower-level components. For example, an abstract component A is composed of the abstract components B, C, and D, with each of them being composed of other components (e.g., the abstract component B is composed of E, F, and G). Assuming that the composition structure is known, our verification approach can be summarized as follows:

1. Perform functional verification of each unit component at the leaf level, say level n. Examples are E, F, G, and H. Let $i = n - 1$.
2. For each abstract component at level i, verify its realization behavior using controlled composition, e.g., the realization behavior of B is the controlled composition of E, F, and G.
3. Generate the specification behavior of each abstract component at level i using synchronized abstraction and projection. The specification behavior is the representative behavior of the abstract component.
4. Repeat from Step 2 with $i = i - 1$ if $i > 0$. In the example, A is the next target abstract component.

Note that this approach systematically generates high-level composition behavior and localizes verification problems. For example, component G is verified as a unit component and used in all three of the abstract components B, C, and D. Once the verification of the controlled composition is done, however, the detailed realization of each abstract component is hidden, leaving only its specification behavior. In this way, the verification of a higher-level abstract component involves only its immediate sub-components, which localizes the verification problem. This approach reduces the complexity of behavior models and systemizes the verification process.

3. Components

In this section, we define the component model used throughout this paper and introduce the notion of *abstract* component, which is the basis for the component-based development and verification methodology named Marmot [14].

3.1. Component model

A component is defined as a typical labeled transition system, but with an emphasis on the role of ports and interfaces.

Definition 1. A component $M = (S, R, N, P, I, E, A, G, Attr, \sum)$ is defined as follows:

- S is a set of states.
- $R \subseteq S \times P \times E \times G \times A \times S$ is a set of transitions.
- N is a set of names.
- $P \subseteq N \times I \times T$ is a set of ports where N is an identifier and $T = \{provide, use\}$ is a type of port. In other words, a port is a named set of events that is either provided or used by the component.
- $I \subseteq 2^E$ is a set of interfaces. In other words, an interface is a set of events.

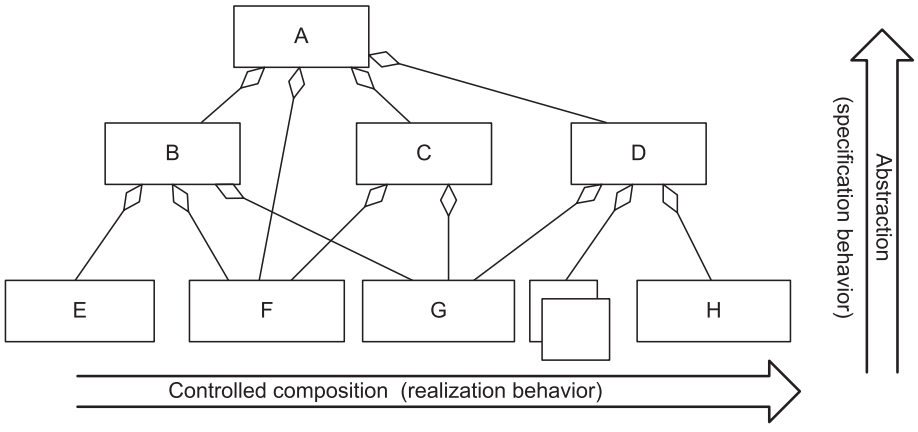


Fig. 4. Checking communication consistency.

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

- E is a set of events. Events can be signals or method calls, and are classified into *sync* events and *async* events.
- $A \subseteq [P] \times \sum$ is a set of pairs consisting of an action and a port. P is optional.
- $G = \{g | g := lval\ [op\ rval], eval(g) \in \{true, false\}\ or\ g := true\}$ is a set of Boolean expressions, where $lval$, $rval \in Attr$ and $op \in \{==, <=, <, >=, >, !=\}$.
- $Attr$ is a set of attributes of the component.
- $\sum$ is a set of actions. Actions can be calling methods, sending signals, or internal actions such as value assignments. $\sum \cap E$ is not necessarily empty.

For example, an action is optionally paired with a port to specify actions as outgoing messages. An attribute, whose type is assumed to be one of the basic data types, such as *bool*, *int*, *char*, and *string*, holds representative values of the component. We note that the definition emphasizes the role of communication ports. For example, component A in Fig. 2 contains two "use" ports named b and d (denoted with a circled end), and two "provide" ports named a and c (denoted with a half-circled end). Though the details are not shown in the figure to save space, port c supports the interface *Init* and d supports the interface *Alarm*, which is a set of operation signatures and/or signals.

$$\{void\ start(unsigned\ dt),\ void\ stop(),\ void\ fired(),\ bool\ isRunning,\ void\ startAt(unsigned\ t0, unsigned\ dt),\ unsigned\ getAlarm(),\ unsigned\ getNow()\}.$$

The port with the "provide" type publishes services provided by the component and the port with the "use" type declares the services the component requires. Several ports can support the same interface in one component, with each port being uniquely identified with its name and interface. It is assumed that each port is bidirectional for sending and receiving messages.

According to the notation explained above, the set of ports of component A in Fig. 2 can be specified as

$$P = \{(a, Init, provide), (c, Alarm, provide), (b, Msp430Timer, use), (d, Msp430TimerControl, use)\}$$

Actions are classified into three types; For each action $a \in \sum$, we denote the type of action as $a.type \in \{sync, async, internal\}$, where *sync* represents synchronous method calls, *async* represents *asynchronous* event sending, and *internal* represents internal value assignments. Events are classified into two types, synchronous events and asynchronous events; synchronous events require an explicit return of the result after the event processing.

A transition $r = (s, p, e, g, a, t) \in R$ specifies that the current state $s \in S$ transits to the next state $t \in S$ after performing the action $a \in A$ when an event $e \in E$ is received at port $p \in P$ and the guarding condition g is evaluated as true. For notational convenience, a transition $r = (s, p, e, g, a, t)$ is also alternatively written as $r : s \xrightarrow{p?e[g]/a} t$. Here, s and t are called the source state and the target state of the transition r, respectively. We denote them with $source(r) = s$ and $target(r) = t$. A trace of an abstract component is a sequence of transitions $r_1 r_2 r_3 \ldots r_k \ldots$, where the target state of r_i is the source state of r_{i+1}.

3.2. Abstract component

An abstract component [14] is a conceptual functional unit of a system that can be executed independently when it is realized. An abstract component is a black box with explicitly specified and externally visible behavior that acts as a contract. This external behavior can be realized internally and independently from other components, or it can use the functionality of other abstract components. The process of realizing an abstract component may involve decomposition or reuse of existing components, depending on whether the realization requires defining a new abstract component or whether it can reuse functionality of existing abstract components. Therefore, each abstract component is a root of a decomposition tree whose leaves are physical components. An abstract component is said to be physically implemented if all of its leaf components are concretized. A service of an abstract component can only be accessed through its port, which provides the specific service.

An abstract component can be specified in terms of the component model defined in the previous section as follows:

Definition 2. A port binding map $BMap$: $P_1 \rightarrow P_2$ is a map binding a port in P_1 to a port in P_2 with the same type and the same interface, i.e., $BMap = \{(p_1, p_2) | p_1 = (n, i, t) \in P_1,\ p_2 = (n', i', t') \in P_2\ with\ i = i' \wedge t = t'\}$.

Definition 3. An abstract component M_{abs} consists of a specification component $M_{spec} = (S, R, N, P, I, E, A, G, Attr, \sum)$ and a set of realization components $M_{real} = \{M_i = (S_i, R_i, N_i, P_i, I_i, E_i, A_i, G_i, Attr_i, \sum_i)\}_{i \in \mathcal{N}}$ that satisfies the following conditions:

- $\exists$ a port binding map $BMap : P \rightarrow \bigcup_{i \in \mathcal{N}} P_i$ that maps each port in P to a port in $\bigcup_{i \in \mathcal{N}} P_i$.
- $I \subseteq \bigcup_{i \in \mathcal{N}} I_i, E \subseteq \bigcup_{i \in \mathcal{N}} E_i, A \subseteq \bigcup_{i \in \mathcal{N}} A_i$, and $Attr \subseteq \bigcup_{i \in \mathcal{N}} Attr_i$
- For each $r = (s, p, e, g, a, t) \in R$, there exists $p' \in \bigcup_{i \in \mathcal{N}} P_i$ and $r' \in R_i$ such that $BMap(p) = p'$ and $r' = (s', p', e, g, a', t')$.

According to the definitions, the set of components $\{A, B, A\|B\}$ in Fig. 2 can be considered as an abstract component, $A\|B$ being a specification component and B, C being its realization components with a binding map $BMap(a) = a$. The transition relation R is defined in such a way that each triggering event that appeared in the specification component is also a triggering event for a transition in one of the realization components, i.e., all the published services are realized internally.

Note that an abstract component can be a physical component if the set of realization components is empty. We also note that an abstract component may be recursively defined, since the realization components themselves can also be abstract components.

For notational convenience, we represent an abstract component only with its specification component throughout this paper if it is not necessary to consider its realization details. Therefore, abstract components and their specification components are used interchangeably from now on.

4. Controlled composition

Controlled composition constrains the possible sequence of transitions depending on the type of components (active or passive) and on the type of message passing mechanism (synchronous or asynchronous). We define operational models for controlled composition using the following definitions:

Definition 4. A set of states in an abstract component is a union of a set of active states S_a and a set of passive states S_p, where $S_a \cap S_p = \emptyset$.

1. An active state is a state where all outgoing transitions are enabled without receiving an external activation signal.

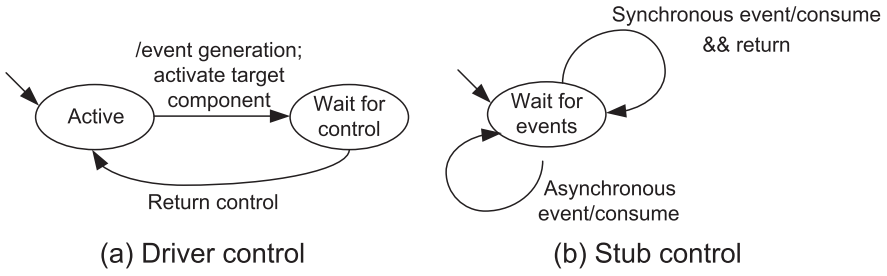


Fig. 5. Control models of drivers and stubs.

2. A passive state is a state where all outgoing transitions are enabled only when an external activation signal is received.

An abstract component can be classified into two types: an active component and a passive component.

Definition 5. Classification of abstract components

1. An active component is a component where all initial states are active states and for any trace π from a passive state s_p, there exists an active state s_a in the trace.
2. A passive component is a component where all initial states are passive states and for any trace π from an active state s_a, there exists a passive state s_p in the trace.

For active (passive) components, there is no cyclic transition that makes the component stay in the passive (active) state indefinitely. An active component reacts to its environmental stimuli actively and independent of other components. Fig. 5 shows representative examples of active and passive components; the environment of a component actively generates events, while stubs of a component passively process external messages.

Definition 6. Operational model

1. An operational model for asynchronous message passing is defined by the rules in Fig. 6.
2. An operational model for synchronous message passing is defined by the rules in Fig. 7.

In Figs. 6 and 7, the predicates above the bar indicate typical transitions in the behavioral model and the predicates below the bar indicate possible transitions in the controlled model, induced from the conditions above the bar.[2] For example, Rule 1 says that if there is a transition $s \xrightarrow{p?e[g]/a} t$ where e is an asynchronous event, a is an asynchronous call action, and s is an active state, then the transition $s \xrightarrow{p?e[g\wedge active]/a} t$ belongs to the controlled model. This means that asynchronous call actions are maintained as they are except for adding a guarding condition to check whether the source state is currently active or not. This is necessary since previous synchronous call actions may have deactivated the active state temporarily. Rule 2 specifies the same case for a passive source state. In this case, the resulting transition requires receiving the *activate* message along with the triggering event to activate a currently passive state. Transitions with internal actions only are treated in a similar manner. On the other hand, transitions with synchronous call actions are changed so that the synchronous call action is concatenated with the *activate* signal that activates the receiver of the message and the *active = false* action that deactivates the currently active configuration (Rule 5 and Rule 6). Rule 7 and Rule 8 are for those cases where the incoming event is a synchronous call and the corresponding action returns the result through the same port. These cases require activating the caller, and, thus, the transition action is concatenated with the *activate* signal.

$$\frac{s \xrightarrow{p?e[g]/a} t,\ e.type = async,\ a = p'!x,\ x.type = async}{s \xrightarrow{p?e[g\wedge active]/a} t}\ s \in S_a \quad (1)$$

$$\frac{s \xrightarrow{p?e[g]/a} t,\ e.type = async,\ a = p'!x,\ x.type = async}{s \xrightarrow{p?e,activate[g]/a} t}\ s \in S_p \quad (2)$$

$$\frac{s \xrightarrow{p?e[g]/a} t,\ e.type = async,\ a \in internal}{s \xrightarrow{p?e[g\wedge active]/a} t}\ s \in S_a \quad (3)$$

$$\frac{s \xrightarrow{p?e[g]/a} t,\ e.type = async,\ a \in internal}{s \xrightarrow{p?e,activate[g]/a} t}\ s \in S_p \quad (4)$$

Fig. 6. Operational rules for asynchronous message passing.

$$\frac{s \xrightarrow{p?e[g]/a} t,\ a = p'!x,\ x.type = sync}{s \xrightarrow{p?e[g\wedge active]/p'!x,activate;active=false} t}\ s \in S_a \quad (5)$$

$$\frac{s \xrightarrow{p?e[g]/a} t,\ a = p'!x,\ x.type = sync}{s \xrightarrow{p?e,activate[g]/p'!x,activate;active=false} t}\ s \in S_p \quad (6)$$

$$\frac{s \xrightarrow{p?e[g]/a} t,\ e.type = sync,\ a = p!x,\ x.type = return}{s \xrightarrow{p?e[g\wedge active]/p!x,activate;} t}\ s \in S_a \quad (7)$$

$$\frac{s \xrightarrow{p?e[g]/a} t,\ e.type = sync,\ a = p!x,\ x.type = return}{s \xrightarrow{p?e,activate[g]/p!x,activate;} t}\ s \in S_p \quad (8)$$

Fig. 7. Operational rules for synchronous message passing.

We note that a triggering event may cause a sequence of actions, i.e., $s \xrightarrow{p?e[g]/a_1;a_2;\ldots a_n} t$. Such cases can be covered by generalizing the operational rules. For example, Rule 5 can be generalized as

$$\frac{s \xrightarrow{p?e[g]/a_1;a_2;\ldots a_n} t,\ \exists i, a_i = p'!x,\ x.type = sync}{s \xrightarrow{p?e[g\wedge active]/a_1;a_2;\ldots p'!x,activate;\ldots a_n;\ active=false} t} s \in S_a.$$

A controlled composition is defined w.r.t. this operational model.

Definition 7. A controlled composition of two components $M = (S, R, N, P, I, E, G, A, Attr, \sum)$ and $M' = (S', R', N', P', I', E', G', A', Attr', \sum')$ is an abstract component $M \underset{\sim}{|} M' = (\widetilde{S}, \widetilde{R}, \widetilde{N}, \widetilde{P}, \widetilde{I}, \widetilde{E}, \widetilde{G}, \widetilde{A}, \widetilde{Attr}, \widetilde{\sum})$, where $\widetilde{S} \subseteq S \times S'$, $\widetilde{N} = N \cup N'$, $\widetilde{P} = P \cup P'$, $\widetilde{I} = I \cup I'$, $\widetilde{E} = E \cup E'$, $\widetilde{G} = G \cup G'$, $\widetilde{A} = A \cup A'$, $\widetilde{Attr} = Attr \cup Attr'$, and $\widetilde{R} \subseteq R \times R'$ is defined by the rules in Fig. 8.

Rules 9–12 are slight variations of typical parallel composition; transitions in each component are preserved with interleavings.

[2] These rules are mainly for receiving messages. Rules for message sending actions are excluded since they only require adding a guarding condition [active] for each sending action without triggering events.

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

$$\frac{s \xrightarrow{p?e[g\wedge active]/a} t\ in\ R,\ s' \xrightarrow{p'?e'[g'\wedge active]/a'} t'\ in\ R'}{(s,s') \xrightarrow{p?e[g\wedge active]/a} (t,s')\ in\ \tilde{R},\ (s,s') \xrightarrow{p'?e'[g'\wedge active]/a'} (s,t')\ in\ \tilde{R}} \quad (9)$$

$$\frac{s \xrightarrow{p?e[g\wedge active]/a} t\ in\ R,\ s' \xrightarrow{p'?e',activate[g']/a'} t'\ in\ R'}{(s,s') \xrightarrow{p?e[g\wedge active]/a} (t,s')\ in\ \tilde{R},\ (s,s') \xrightarrow{p'?e',activate[g']/a'} (s,t')\ in\ \tilde{R}} \quad (10)$$

$$\frac{s \xrightarrow{p?e,activate[g]/a} t\ in\ R,\ s' \xrightarrow{p'?e',activate[g']/a'} t'\ in\ R'}{(s,s') \xrightarrow{p?e,activate[g]/a} (t,s')\ in\ \tilde{R},\ (s,s') \xrightarrow{p'?e',activate[g']/a'} (s,t')\ in\ \tilde{R}} \quad (11)$$

$$\frac{s \xrightarrow{p?e,activate[g]/a} t\ in\ R,\ s' \xrightarrow{p'?e'[g'\wedge active]/a'} t'\ in\ R'}{(s,s') \xrightarrow{p?e,activate[g]/a} (t,s')\ in\ \tilde{R},\ (s,s') \xrightarrow{p'?e'[g'\wedge active]/a'} (s,t')\ in\ \tilde{R}} \quad (12)$$

$$\frac{s \xrightarrow{/a} t\ in\ R,\ s' \xrightarrow{p'?e'[g'\wedge active]/a'} t'\ in\ R'}{(s,s') \xrightarrow{/a} (t,s') \xrightarrow{p'?e'[g'\wedge active]/a'} (t,t')\ in\ \tilde{R}} \quad (13)$$

$$\frac{s \xrightarrow{p?e[g\wedge active]/a} t\ in\ R,\ s' \xrightarrow{/a'} t'\ in\ R'}{(s,s') \xrightarrow{/a'} (s,t') \xrightarrow{p?e[g\wedge active]/a} (t,t')\ in\ \tilde{R}} \quad (14)$$

Fig. 8. Rules for controlled composition.

Rule 13 and Rule 14 specify that active transitions without triggering events have priority over transitions caused by external events.

5. Abstraction

As will be shown in our experiments, controlled composition greatly helps to reduce the complexity of verification, but only up to a certain point. Abstractions are effective techniques for further reducing the complexity of models, and thus their verification costs. In this section, we introduce two abstraction techniques specialized for abstracting internal communications and unnecessary services for a given abstract component: Synchronized abstraction simplifies detailed communication behavior among abstract components by removing unnecessary information w.r.t. the functionality of the component. Projection abstraction extracts a higher-level abstract behavior relevant to its published services defined in its ports.

5.1. Synchronized abstraction

Synchronized abstraction is used to compose abstract components into one higher-level abstract component, based on the dependencies specified in the port connection. The idea is illustrated in Fig. 9. In Fig. 9a, *A* and *B* are two abstract components where the *use* port *b* of *A* is connected to the *provide* port *c* of *B*. It is assumed that the two ports *b* and *c* support the same interface.

The external behavior of the two abstract components and their parallel composition are illustrated as statemachines in Fig. 9a and b, respectively. Fig. 9c shows the reduction of the parallel composition using the port dependency; the port connection can be abstracted by replacing the name of the provide port with the name of the use port, e.g., *c* with *b*, resulting in a natural reduction of transitions. After the reduction, the transition from the state labeled (1,3) and the state (1,4) is removed because the composition of *A* and *B* results in hiding port *b* of *A*. Fig. 9d is the result of removing the unreachable state (1,4) from the initial state (1,3).

Definition 8. A synchronized reduction of an abstract component $M = (S, R, N, P, I, E, G, A, Attr, \sum)$ w.r.t a pair of dependent ports p_1,$p_2 \in P$, where p_1 is a use port and p_2 is a provide port, is an abstract component $M_{sync}(p_1 \to p_2) = \left(\widehat{S}, \widehat{R}, N, \widehat{P}, I, E, G, A, Attr, \sum\right)$, where $\widehat{S} \subseteq S, \widehat{P} = P - \{p_1, p_2\}$, and $\widehat{R} \subseteq R$ is recursively constructed by applying the following rules:

1. Initially, $\forall s \in S, s \in \widehat{S}$.
2. For each consecutive transition $r_i : s_i \xrightarrow{p?e[g]/a} s_{i+1}$ and $r_{i+1} : s_{i+1} \xrightarrow{p'?e'[g']/a'} s_{i+2}$ in R,

(a) if $p = p_1$, $a = \langle \tilde{a}_i \rangle\ p_2!x$, $x = e'$, and $p' = p_2$, where $\langle \tilde{a}_i \rangle$ is a sequence of assignment actions, then $r_i, r_{i+1} \notin \widehat{R}$ and $r'_i : s_i \xrightarrow{p?e[g\wedge g']/\langle \tilde{a}_i \rangle a'} s_{i+2}$ is in $\widehat{R}$.

(b) if $p = p_1$, $a = p_2!x$; $\langle \tilde{a}_i \rangle$, $x = e'$, and $p' = p_2$, where $\langle \tilde{a}_i \rangle$ is a sequence of assignment actions, then $r_i, r_{i+1} \notin \widehat{R}$ and $r'_i : s_i \xrightarrow{p?e[g\wedge g']/a'\langle \tilde{a}_i \rangle} s_{i+2}$ is in $\widehat{R}$.

(c) Otherwise, $r_i, r_{i+1} \in \widehat{R}$.

3. Remove s from $\widehat{S}$ if there is no incoming transition to s in $\widehat{R}$.

The reduction is repeated for each pair of dependent ports. For example, Fig. 9c shows the result of the synchronized reduction of Fig. 9b. First, the port name *c* is replaced with *b* according to the port connection information. Second, all the consecutive transitions that need communication through the same port are reduced

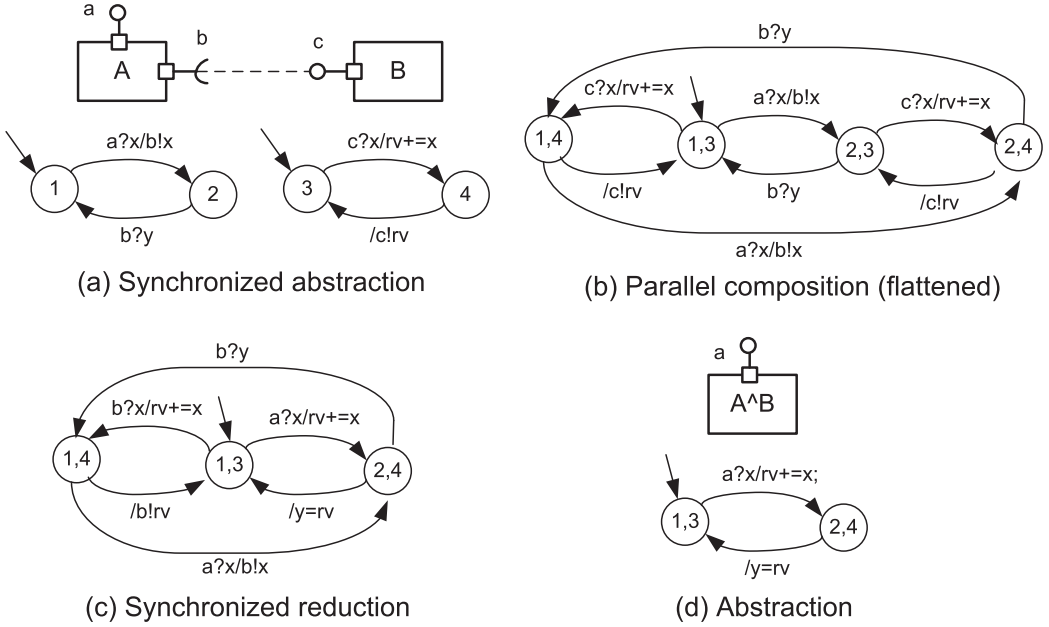


Fig. 9. Synchronized abstraction.

to local actions; e.g., $(1,3) \overset{a?x/b!y}{\rightarrow} (2,3) \overset{b?x/rv+=x}{\rightarrow} (2,4)$ is replaced with $(1,3) \overset{a?x/rv+=x}{\rightarrow} (2,4)$ and $(2,4) \overset{/b!rv}{\rightarrow} (2,3) \overset{b?y}{\rightarrow} (1,3)$is replaced with $(2,4) \overset{/y=rv}{\rightarrow} (1,3)$.

This process hides communications among internal components that comprise an abstract component, replacing internal message passings with local actions. Note that each pair of connected ports is now hidden from the external view of the newly constructed abstract component.

After the synchronized reduction, all the transitions triggered by events through the hidden ports can be removed because they are impossible transitions. For example, the transitions $(1,3) \overset{b?x/rv+=x}{\rightarrow} (1,4)$ and $(2,4) \overset{b?y}{\rightarrow} (1,4)$ are not possible because *b* is a use port that sends a message first and receives its return value; since sending a message through port *b* has already been reduced, there cannot be any more receiving messages through that port. After removing these transitions from (c), the state *(1,4)* is also removed because it does not have any incoming transitions anymore. As a consequence, the transition $(1,4) \overset{a?x/b!x}{\rightarrow} (2,4)$ is also removed. The synchronized abstraction is finalized by removing all those transitions and states that are unreachable from the initial states.

Definition 9. A synchronized abstraction of a component $M = (S, R, N, P, I, E, G, A, Attr, \sum)$ w.r.t. a hidden port $p_h \notin P$, denoted as $M_{abs}(p_h) = \left(\widehat{S}, \widehat{R}, N, P, I, E, G, A, Attr, \sum\right)$, is a synchronized reduction of *M* with the following rules:

$$\frac{s, s' \in S,\ s \overset{p?e[g]/a}{\rightarrow} s' \text{ in } R, p \neq p_h}{s, s' \in \widehat{S}, (s \overset{p?e[g]/a}{\rightarrow} s') \in \widehat{R}} \quad (15)$$

$$\forall s \in \widehat{S}, \exists r \in \widehat{R},\ \text{such that } s = target(r) \quad (16)$$

The rule says that all the transitions triggered through hidden ports do not belong to the abstract behavior model, while transitions triggered through other ports are preserved, and all the states in the abstract model have incoming transitions into the states after abstraction.

5.2. Projection abstraction

In embedded systems development, hardware components are often assembled into one abstract component that represents a collective functionality. Oftentimes, the sub-components assembled to form the abstract component contain implementation details that are unnecessary for the abstract component. For example, the *Msp*430*TimerC* component in our case example TinyOS [40] contains dozens of ports supporting 10 virtual timers. On the other hand, *Msp*430*Timer*32*khzC*, an abstraction of *Msp*430*TimerC*, supports only three ports that are necessary for providing a timer, and, thus, the behaviors related to the seven other timers are irrelevant to the behavior of *Msp*430*Timer*32*khzC*. Projection abstraction is used to abstract out such irrelevant behaviors after synchronized abstraction has been applied. These abstractions are based on a port binding map that associates a pair of ports with the same interface type and the same port type.

Definition 10. Projection abstraction: Given an abstract component $M = (S, R, N, P, I, E, G, A, Attr, \sum)$, a set of ports P_{abs}, and a port binding map $BMap$: $P_{abs} \rightarrow P$, the projection abstraction $M(BMap) = \left(\widehat{S}, \widehat{R}, \widehat{N}, P_{abs}, \widehat{I}, \widehat{E}, G, A, Attr, \sum\right)$ is an abstract component where the triggering events are restricted to only those events supported by ports in P_{abs}.

1. $\widehat{N}$ is a set of port names for P_{abs}.
2. $\widehat{I} = \{i \in I | \exists p \in P_{abs}, n \in \widehat{N}, t \in T_{in}$ such that $p = (n, i, t)\}$, i.e., a set of interfaces that are supported by the ports in the abstract component.
3. $\widehat{E} = \{e \in E | \exists p \in P_{abs}, n \in \widehat{N}, t \in T_{in}$ such that $p = (n, i, t) \wedge e \in i\}$, i.e., a set of events that are supported by the ports in the abstract component.
4. $\widehat{S} = \{s, t \in S | r = (s, p, e, a, t) \in R \ \wedge \ p \in BMap(P_{abs}) \wedge e \in \widehat{E}\}$, i.e., all the source and target states in a transition in *R* whose triggering events are supported by P_{abs} belong to $\widehat{S}$.
5. $\widehat{R} = \{r \in R | r = (s, p, e, a, t) \ \wedge \ p \in BMap(P_{abs}) \wedge e \in \widehat{E}\}$, i.e., all the transitions in *R* with the triggering events supported by P_{abs} belong to $\widehat{R}$.

Fig. 10 illustrates a simple example of a projection abstraction: A component *A* with two provided ports named *IO*1 and *IO*2 is abstracted to a component with one provided port named *Led*0, where *IO*1, *IO*2, and *Led*0 all support the same interface type *Led* and *Bmap* assigns the abstract port named *Led*0 to *IO*1.

6. Verification

The proposed controlled composition and abstraction techniques play a key role in our systematic compositional verification approach illustrated in Fig. 11. For a given unit component, a unit verification model consists of the target component behavior, the drivers derived from its *provide* ports, and stubs derived from its *use* ports. This unit verification model is translated into PROMELA, the input language of SPIN, by translating each model as independent processes that communicate with each other only through port connections. Using the model checker SPIN, we can comprehensively

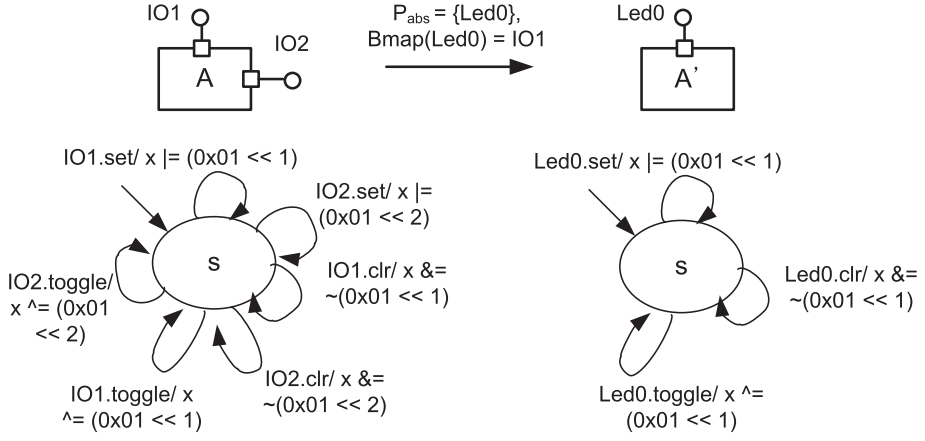


Fig. 10. Projection abstraction.

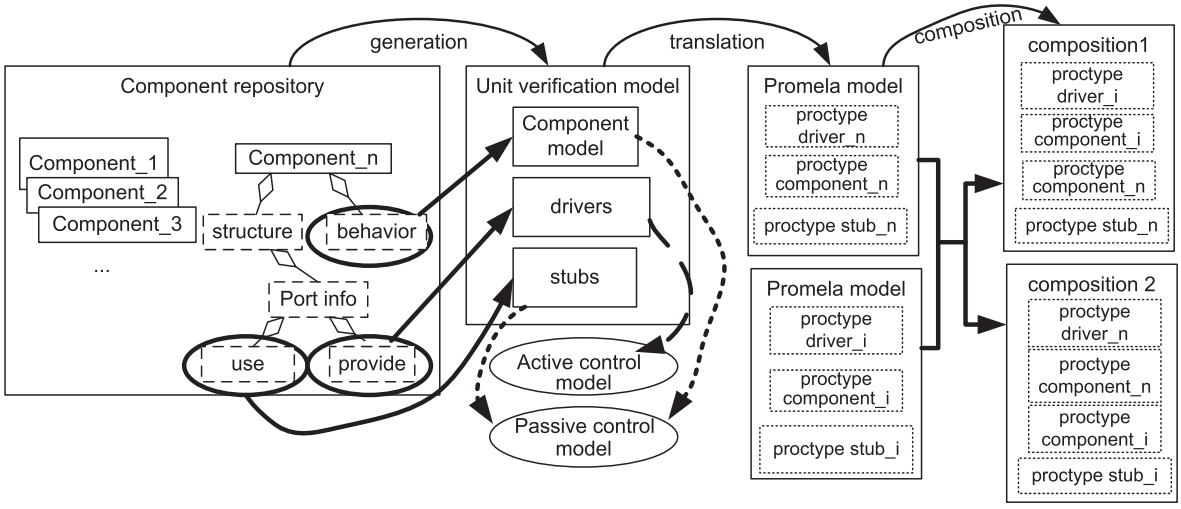


Fig. 11. Verification framework.

test the existence of process deadlock and/or incorrect call sequences of the unit component. The verification of composite components is performed by simply replacing drivers and/or stubs of a unit component with the actual components.

In this verification framework, all the structural and behavioral specifications for each physical component as well as those of abstract components are assumed to be maintained in a component repository. Therefore, abstractions are applied in the construction of abstract components rather than during verification. Controlled composition is applied during the translation of the unit/composite verification model. This translation process is automated for the abstract components specified in UML2.0 notation as an extension of the Marmot verification framework [14].

6.1. Generation of verification model

Each verification model consists of three elements: (1) one or more target components, (2) a mock component acting as a driver, and (3) mock components acting as stubs. By default, the target components and the stubs are passive components unless specified otherwise.[3] The driver is an active component that continuously generates external stimuli to the target component. These three elements are generated from the structural and behavioral models of the target unit component.

Given a unit component, a driver is generated from the information on its *provide* ports; a *provide* port specifies the services provided by the component and the component is designed to react to external service calls. Therefore, the probable user of the component, i.e., the driver, is supposed to generate service calls specified in the "provide" port. The default driver generates events and service calls randomly and infinitely.

For example, if an abstract component A has two *provide* ports p_0 and p_1, where p_0 supports {init} and p_1 provides {start, stop, get}, a driver continuously generates the *init* event and the other driver continuously generates one of the {start, stop, get} events. Fig. 12a illustrates the behavior of the driver that generates events for p_0 assuming that all the operations require a synchronous call. In the figure, any of the three transitions, from s_0 to s_1, from s'_0 to s'_1, and from s''_0 to s''_1, is possible in the initial state; one of them is randomly selected. Since the driver is an active component, the guarding condition [*active*] for the transitions is initially true. After the first transition, say, the transition from s_0 to s_1, all other initially active states become deactivated temporarily until the return message arrives for the synchronous call message.

On the other hand, a default stub is generated from the information on each *use* port. Recall that the *use* port specifies the signatures of services regardless of which component provides them. A default stub acts as a component that provides services without real implementation; it just consumes the request message and returns a random value corresponding to the return type. Each default stub is also considered as a passive component that waits for external activation, processes a request after it is activated, and returns to the passive state after processing the request. For example, if the abstract component has one *use* port p_3, where p_3 supports two operations {set, reset}, then a default stub is generated as a passive component that receives external calls. Fig. 12b shows the behavior of the default stub that consumes messages from *port*3 (connected to p_3), assuming that all the messages are synchronous calls.

In this way, both drivers and stubs are over-approximated, allowing all possible communication behaviors.

6.2. Verification of a unit component

The three elements of the unit verification model are composed using controlled composition. The purpose of unit verification is to ensure the correctness of the component behavior, which can be independently verified regardless of the detailed behavior of its service providers. For example, we can verify the consistency of the call signature, the correctness of call sequences in the component, and the freedom from process deadlock. Due to its use of formal language, model checking can also identify an incorrect number of parameters in message sending, which might not be detected by simulation or testing.

Note that any verification activity at this stage may detect only relatively simple errors and generates false negatives since the verification environment is not accurately modeled. Nevertheless, the effectiveness of model checking increases as drivers and stubs are incrementally replaced with actual components.

6.3. Incremental composition and verification

Two unit components can be composed by connecting their ports according to the following port connection rules:

[3] Hardware components are generally active components.

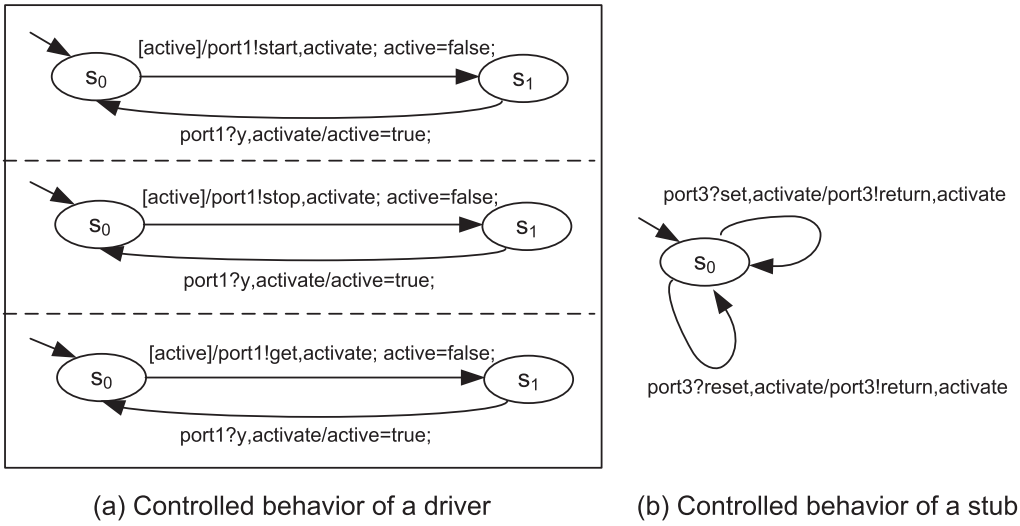


Fig. 12. Verification environment.

Definition 11. Port connection rules. Two ports p_1 of component C_1 and p_2 of component C_2 can be connected if and only if

1. The two ports support the same interface, i.e., $p_1|_I = p_2|_I$.
2. The two ports are of different types, i.e., $p_1|_T \neq p_2|_T$.

For example, port *b* of component *A* in Fig. 2 can be connected to port *e* of component *B* because they support the same interface, but with different port types; *b* is the *use* type, whereas *e* is the *provide* type.

Note that composition in our context is a simple replacement of drivers or stubs with actual components. Given two unit components C_1 and C_2, some possible compositions are:

1. If a *use* port p_1 of C_1 is connected to a *provide* port of C_2, then C_2 replaces the stub that was connected to p_1.
2. If a *provide* port p_1 of C_1 is connected to a *use* port of C_2, then C_2 replaces the driver that was connected to p_1.

This is a systematic process and no extra work is required to modify the control model. As components are incrementally composed, we can perform verification with more realistic models.

6.4. Verification problems

The abstraction techniques hide communication details focusing on the functional behavior, systematically generating high-level behavior models of abstract components. Nevertheless, we first need to verify interaction consistency among components before applying abstractions to assure that the abstraction does not result in overlooking communication-related problems. This is called *horizontal verification*. We also show that the specification behavior constructed by synchronous abstraction is a sound abstraction of the realization behavior. This is called *vertical verification*.

6.4.1. Horizontal verification

When composing several abstract components, it is important to assure that the communication behavior is safely defined, without any possibility of system failures caused by communication mismatches. For example, a component may send a synchronous call to another component and wait for the return while the other component is busy processing other signals and ignores the call. This may result in a communication deadlock situation.

Definition 12. Communication consistency. Let M be a controlled composition of two abstract components M_1 and M_2, and let $Dmap: P_1 \rightarrow P_2$ be a dependency relation between ports in M_1 and M_2. Then M (with all dependent ports connected) is said to be consistent in its communication behavior if it runs infinitely under the infinite sequence of events generated from its drivers through external ports.

This communication consistency can be checked using model checking [17,33]. For example, the SPIN model checker [33] is equipped with a verification option – invalid end-state checking – that can be used to check communication consistency.

6.4.2. Vertical verification

Once horizontal verification succeeds, the next step is to verify that the specification model is an abstract representation of the realization model. We prove this by using the notion of trace refinement; M' is a trace refinement of M w.r.t. a set of ports P_{abs} if for all traces of M triggered by an external event through P_{abs}, there is an equivalent trace in M' processing the same external event with the same action.

Definition 13. Given two components $M = (S, R, N, P, I, E, G, A, Attr, \sum)$ and $M' = (S', R', N', P', I', E', G', A', Attr', \sum'), M'$ is a trace refinement of M with respect to a set of ports P_{abs} if the following is satisfied:

- For any transition $r = (s, p, e, g, a, t)$ in M, where $p \in P_{abs}$, there exists a finite trace $r'_1 r'_2 r'_3 \ldots r'_k$ in M', where $r'_i = (s'_i, p'_i, e'_i, g'_i, a'_i, t'_i)$ such that
  1. $s = s'_1$,
  2. $p = p'_1$,
  3. $e = e'_1$,
  4. $\forall i \in \{1, \ldots, k\},\ g \Rightarrow g'_i$,
  5. $a = \langle a'_j \rangle$, where a'_j represents an action in $\{a'_1, a'_2, \ldots, a'_k\}$ and $\langle a'_j \rangle$ represents a sequence of actions from $\{a'_1, a'_2, \ldots, a'_k\}$,
  6. $attr(s) = attr(s'_1)$ and $attr(t) = attr(s'_k)$. Here, $attr(s)$ denotes a set of attribute values on s.

Theorem 1. *If $M = (S, R, N, P, I, E, G, A, Attr, \sum)$ is a synchronized abstraction of $M' = (S', R', N', P', I', E', G', A', Attr', \sum')$, then M' is a trace refinement of M with respect to P.*

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

Proof. Each $r = (s,p,e,g,a,t)$ in M has two possible cases; (1) r is the result of the synchronized abstraction of $r'_1 r'_2 r'_3 \dots r'_k$ in M', or (2) $r \in R'$. For case 1, $s = s'_1, p = p'_1, e = e'_1$, and $a = \langle a'_j \rangle$ by the definition of synchronized abstraction. $\forall i \in \{1,\dots,k\},\ g \Rightarrow g'_i$ is true since $g = g'_1 \wedge g'_2 \wedge \dots \wedge g'_k$. $attr(s) = attr(s'_1)$ is true because the abstraction maintains the first state s'_1. $attr(t) = attr(s'_k)$ is also true since all the assignment actions from r'_1 to r'_k are sequentially performed at the incoming transition to s'_k, and, thus, the valuation of attribute values on t is the same as the valuation of attribute values on s'_k. Therefore, $r'_1 r'_2 r'_3 \dots r'_k$ is the finite trace that refines r. For case 2, let $r' = r$. Obviously, r' is the finite trace that refines r. ■

If we consider only the values of attributes for each state, we can define a bisimulation relation between the abstract components before and after synchronized reduction.

Definition 14. Given two components $M = (S,R,N,P,I,E,G,A,Attr,\sum)$ and $M' = (S',R',N',P',I',E',G',A',Attr',\sum')$, a relation $H \subseteq S \times S'$ is defined as $H(s,s')$ if and only if $attr(s) = attr(s')$.

The proof that H defines a bisimulation relation on $S \times S'$ uses the fact that a transition with a sequence of actions can be expressed in a semantically equivalent way by expanding the transition with a sequence of transitions with null triggering events; for example, $s \xrightarrow{p?e[g]/\langle a_i \rangle_{i=1..n}} t$ is equivalent to $s \xrightarrow{p?e[g]/a_1} s_1 \xrightarrow{\tau/a_2} s_2 \xrightarrow{\tau/a_3} s_3 \dots \xrightarrow{\tau/a_n} t$, where τ means the transition can occur without any events or conditions. In order to facilitate a intuitive understanding of the proof, we introduce two auxiliary definitions; for a transition r, $r|_A$ represents the action part of the transition and $r|_S$ represents the target state of the transition, i.e., $r|_A = a$ and $r|_S = t$ for $r = (s,p,e,g,a,t)$.

Theorem 2. *Let M be a synchronized reduction of M'. Then H is a bisimulation relation on $S \times S'$. Thus, M and M' are bisimulation equivalent.*

Proof. We need to prove that (1) $H(s_0, s'_0)$ where s_0 and s'_0 are initial states of S and S', respectively, (2) if $H(s,s')$, then for each next state s_1 of s, there exists a next state s'_1 of s' such that $H(s_1, s'_1)$, and (3) if $H(s,s')$, then for each next state s'_1 of s', there exists a next state s_1 of s such that $H(s_1, s'_1)$.

First, $H(s_0, s'_0)$ holds since the synchronized abstraction reduces only transitions and does not change the initial states.

Second, suppose $H(s,s')$, then each outgoing transition r from s is either the result of synchronized reduction from $r'_1 r'_2 r'_3 \dots r'_k$ or $r \in R'$. For the former case, rephrase $r = s \xrightarrow{p?e[g]/<a_i>_{i=1..n}} t$ as $r_1 r_2 \dots r_k$ where $r_1 = s \xrightarrow{p?e[g]/<a_1 \dots a_{1_j}>} t_1, r_i = t_{i-1} \xrightarrow{\tau/<a_{i-1_j} \dots a_{i_j}>} t_i$ with $i = 2..k, r_i|_A = r'_i|_A$, and $t_k = t$. Note that such a trace $r_1 r_2 \dots r_k$ is unique for each $r'_1 r'_2 r'_3 \dots r'_k$ by the construction of synchronized reduction. Let $s_1 = t_1$ and $s'_1 = r'_1|_S$, then $H(s_1, s'_1)$ holds. $H(s_i, s'_i)$ holds for $i = 2..k$ by induction where $s_i = t_i$ and $s'_i = r'_i|_S$. For the latter case, let $s'_1 = s_1$. Then $H(s_1, s'_1)$ holds and $s'_1 = s_1$ is a next state of s'.

Third, suppose $H(s,s')$, then each outgoing transition r' from s' has two possible cases; r' is a part of a sequence of transitions $r'_1 r'_2 r'_3 \dots r'_k$ in M' which is reduced to $r = s \xrightarrow{p?e[g]/<a_i>_{i=1..n}} t$ in M or $r' \in R$. For the former case, we can find a unique sequence of transitions $r_1 r_2 r_3 \dots r_k$ in M such that $H(r_i|_S, r'_i|_S)$ for each $i = 1\dots k$ with the same argument as in the second proof. Let's say $r' = r'_i$ in this sequence of transitions, then $s_1 = r_i|_S$ satisfies $H(s_1, s'_1)$ for $s'_1 = r'_i|_S$. For the latter case, let $s_1 = r'|_S$, then $H(s_1, s'_1)$ holds and s_1 is a next state of s. ■

Therefore, state properties verified after synchronized abstraction are also valid before the abstraction if we assume that transitions do not cost time. We note that the abstraction does not introduce false negatives for state properties. However, false negatives are possible for path properties such as "X happens at least two transitions after Y becomes true" after the abstraction.

Theorem 3. *Let M be an abstract component after applying controlled composition, synchronized abstraction, and/or projection abstraction on M'. Then any state properties verified on M are valid on M' if the external events are limited to those published in M.*

Proof. It follows from Theorem 2 and the fact that (1) the projection abstraction removes behavior from M' only when they are not triggered by the published external events, and (2) controlled composition removes impossible behaviors on serial machines, but retains all sequential behaviors of M'. ■

7. Experiments

In order to see the effect of the proposed controlled composition and abstraction, we have conducted a series of experiments on a sensor network application running on top of TinyOS.

7.1. TinyOS components

TinyOS [1] is an open-source operating system for wireless sensor networks developed at the University of California at Berkeley. Its core code consists of less than 4000 bytes of codes, which consume less than 256 bytes of data memory. It supports event-based multi-tasking, aiming at low-cost embedded operating systems suitable for embedded networking. The programming language of TinyOS, called nesC [27], is a C dialect with component-based constructs. The size of the program code is relatively small, but it still contains most of the complex behavior of generic operating systems.

Fig. 13 shows the top-level structure of the abstract components of a TinyOS application, RadioCountToLeds, which is released with TinyOS as a case example. The application maintains a 4 Hz counter, broadcasts the value of the counter when it is updated, and displays the bottom three bits of the counter on the LEDs. As shown in Figs. 13 and 14, the application structure including its underlying TinyOS is modeled using the notion of abstract components from the top level to the bottom physical component. At the top level, the application consists of seven abstract components as illustrated in Fig. 13, where each abstract component is labeled in alphabetical order and is refined into an internal structure. Each abstract component at the top level is successively refined; for example, one of the abstract components, *TimerMilliC* (labeled with *D*) is refined into two physical components D_{31}, D_{32}, and one abstract component D_{33}, which is further refined (Fig. 14). Fig. 15 is a simplified internal model of D_{33}.

This structural modeling process is rather straightforward, since the organization of TinyOS components is well matched with our notion of abstract component. However, we need to construct the behavioral model of each abstract component, since it is not explicitly specified in the abstract-level TinyOS components. We reverse-engineered the behavioral model from the bottom-level components by applying composition and abstraction successively.

7.2. Experimental settings

The goal in this series of experiments is to see the effect of controlled composition and abstraction in the verification of TinyOS

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

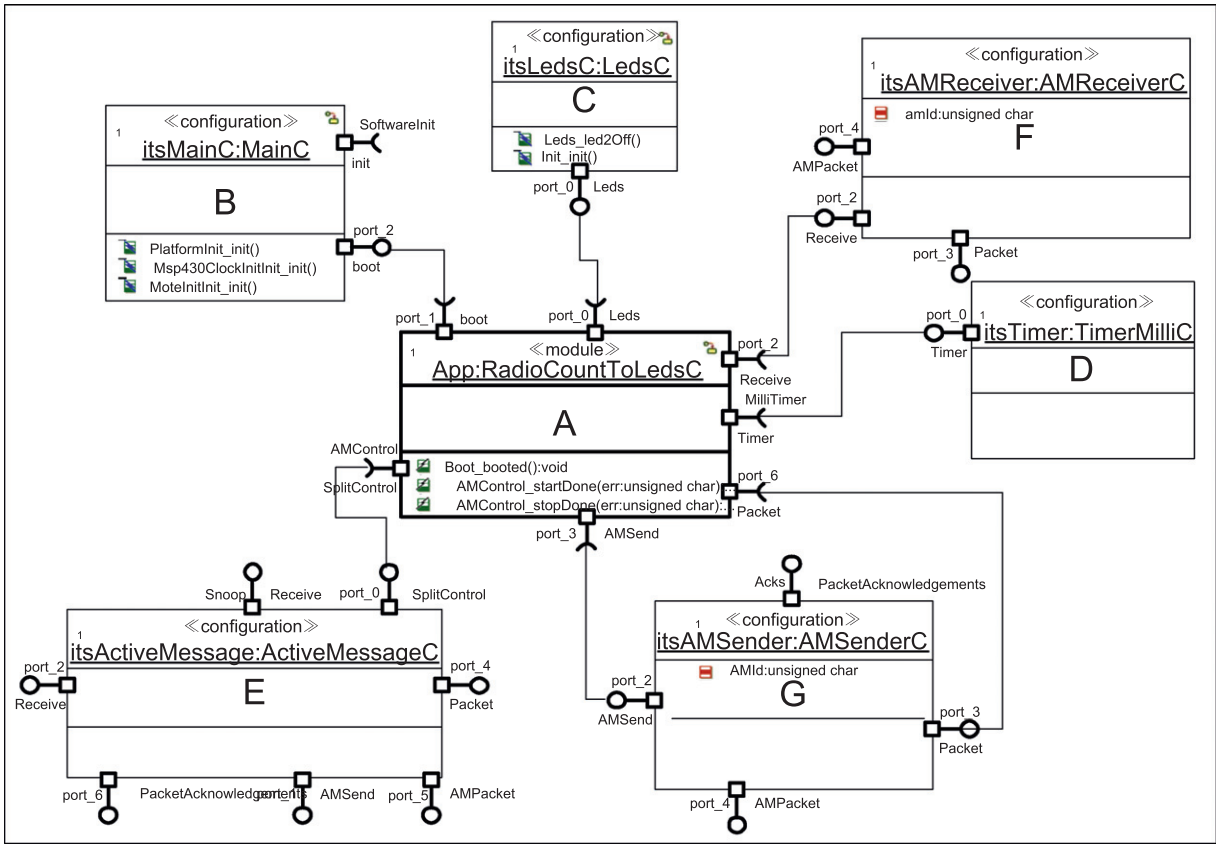


Fig. 13. Top-level structure of a TinyOS application.

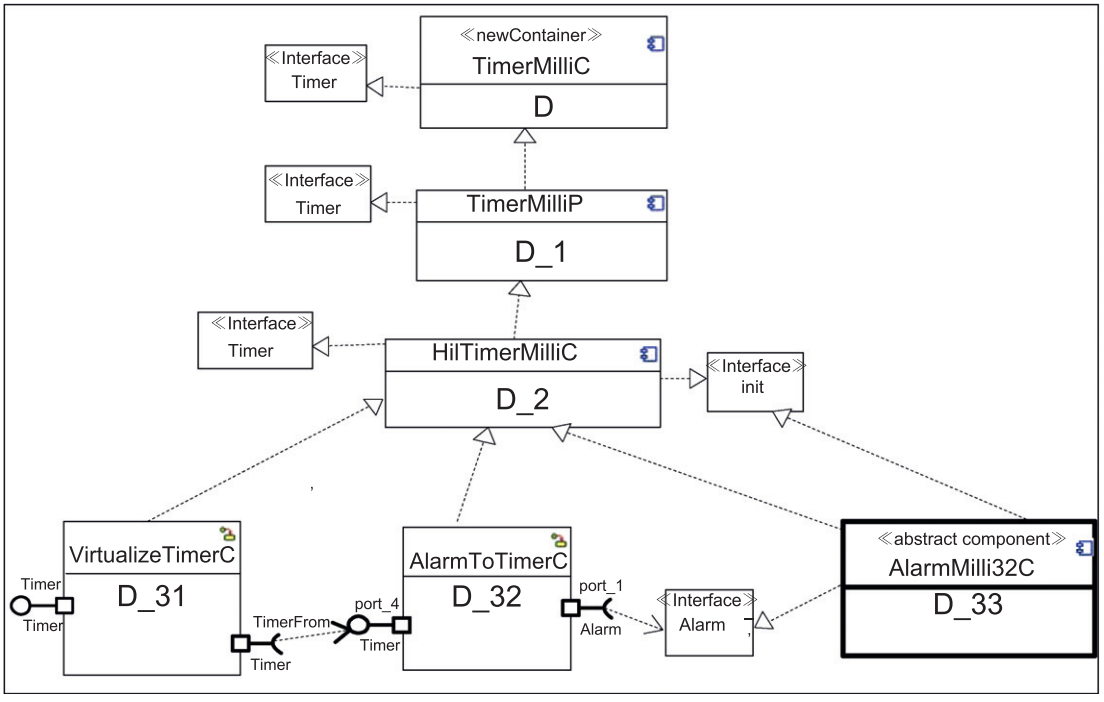


Fig. 14. The internal structure of the abstract component *D*.

components as bottom-up behavioral composition is performed. To compare the performance, we conducted three sets of verifications at the initial step (for the verification of composition behavior at the bottom level) as follows, where the verification purpose of each experiment was to show the communication consistency as stated in Definition 12.

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

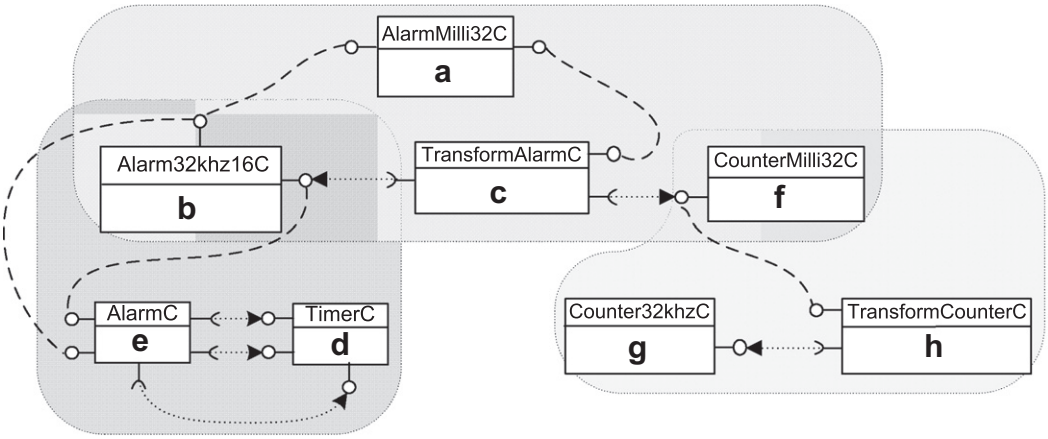


Fig. 15. The internal structure of D_{33}.

1. Composite components are verified with conventional parallel composition and no abstraction is applied to generate specification behavior.
2. Composite components are verified with controlled composition and no abstraction is applied to generate specification behavior.
3. Composite components are verified with controlled composition and specification behavior is generated and used for the verification of the next-level composite components.

The target component for the comparative verification is D_{33} in Fig. 14, whose internal composition structure is shown in Fig. 15. All the experiments were performed on a PC with a 2-Gigahertz Pentium II processor and 2 Gigabytes of memory; the maximum allocable memory during Spin verification was set to one Gigabyte and the verification time threshold was set to 1 h. Two verification options were used in the experiments with Spin: the exhaustive search option by default and the bit-state space hashing option, based on partial search, for the cases when exhaustive search does not scale up. Though verification using the bit-state space hashing option can be unsound, we can judge the verification coverage from the hash factor reported by the model checker.

7.3. Verification with parallel composition

Verification with parallel composition required much effort and a lot of resources. First, it generated a large number of false negatives, reporting unrealistic errors in the communication model. Second, when all the unrealistic errors were finally dismissed, the model checker Spin succeeded in verifying each unit verification model, but failed to verify composite components due to an out-of-memory error – a typical state-space explosion problem in model checking.

A counterexample identified in the verification of component *E* is stated below:

1. The environment of *E* calls the service *start*.
2. The *start* operation calls *getNow* internally.
3. The environment calls the service *start* again, but this call is pending, since *E* is now in the middle of processing the previous call.
4. The service provider of *getNow* sends a return message, but it is pending, since the previous *start* message should be processed first.
5. The environment can send messages until the message buffer becomes full, and eventually, no more progress is possible.

This situation can be an actual process deadlock situation if the service call *start* from its environment is an asynchronous call; the environment may send messages indefinitely without waiting for the result of its prior calls. For the case of synchronous calls, however, this type of counterexample is not possible, since the environment does not send the next message before receiving the result of its prior message. In fact, *start* is defined as a synchronous call, and thus, this counterexample is a false negative.

Table 1 shows the verification results after dismissing all such false negatives. The first column of the table indicates the names of the abstract components in Fig. 15. The second column is for the type of the component, which can be either a unit component (U) or a composite one (C). The performance and the cost of verification are shown in columns three to seven; starting from the third column, the columns represent the search depth, the number of states searched, the number of transitions searched, the amount of memory consumed in Mega bytes, and the amount of time required for the verification in seconds, in that order. The last column indicates the hash factor when the bit-state hashing option was used. The default verification option was exhaustive search.

The verification of unit components performed well with a moderate amount of time and memory. For example, Spin consumed 162.47 Mbytes of memory and 115 s to model-check component *e* exhaustively. However, the verification of composite components performed quite poorly; Spin failed to exhaustively verify *b*, which is the composition of *d* and *e*, exceeding the allowed

Table 1
Experiments on parallel composition.

Name	Type	Depth	States	Transitions	Memory	Time	H/F
e	U	39,146	20,245,676	24,523,053	162.47	115.00	
d	U	8361	162,844	238,969	8.85	0.17	
b (*d* + *e*)	C	OM	OM	OM	OM	OM	
b (*d* + *e*)	C	214,157	26,404,801	40,635,277	523.77	48.10	150.91

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

memory limit. The last row of the table shows the verification performance on *b* when the bit-state hashing option was used. Any further composition with *b* spaced out of memory.

7.4. Verification with controlled composition

Table 2 shows the experiment results of the verification using controlled composition. The use of controlled composition reduced verification time and memory for each unit component, but a more drastic improvement can be seen in the verification of compositions. For example, when controlled composition was used for *b*, which is a composition of *d* and *e*, the search depth required for the verification decreased to about one tenth of the search depth required for parallel composition. The number of states and the number of transitions required for the verification decreased to about one fourth, respectively half, of those for parallel composition. Due to the successive reduction in verification complexity, SPIN was able to verify all unit component models and their compositions, except for the top-level component *a*, which is, in fact, a composition of seven components. SPIN failed to verify *a* exhaustively with 1 Gigabyte of memory. It required over 1 h 38 min using the bit-state hashing option, and its expected coverage was only 2.08%. Therefore, we conclude that SPIN fails to scale up to *a*.

A successive composition without abstraction is subject to state-space explosion in the end.

7.5. Verification with controlled composition and abstraction

In the third experiment, each verified composite component was abstracted using synchronized abstraction and projection abstraction. Table 3 shows the result; the verification result of each unit component is the same as that of the second experiment, since both used controlled composition for unit verification models. Each composite component, however, was abstracted after controlled composition and verification. For example, the row for *b_real* in Table 3 shows the verification result of *b* using controlled composition only, whereas the row for *b_spec* shows the verification result after applying abstractions on *b_real*, which reveals a drastic improvement in performance. Similarly, *f_spec* is the verification result after applying abstractions on *f_real*, which is a controlled composition of *g* and *h*. *c_f_real* is a controlled composition of *c* and *f_spec* and *c_f_spec* is its abstracted version. In this way, SPIN succeeded in verifying all composite components up to *a*.

When an abstract component is composed of a number of complex components, it may be impossible to verify the realization component using exhaustive verification methods. For example, *c_f_real* and *a_real* required more than 1 Gbyte of memory for exhaustive verification. In such a case, the bit-state hashing option is used to save verification time and cost, trading the exhaustiveness of verification for efficiency. The results could be unsound, but their hash factors reported by the SPIN verifier show high expected state coverage: over 79% for *c_f_real* and 103% for *a_real*, respectively.

Fig. 16 compares the scalability of memory consumption for the second and the third experiments. We note that memory consumption for compositions with a control model and abstraction grows linearly if we consider only the specification models.

7.6. Verification up to the application component

Once the effectiveness of controlled composition and abstractions had been shown through the comparative experiments up to the abstract component D_{33}, we continued the same controlled composition and abstraction process up to the top-level application component *A*. Table 4 shows the result; D_{31}, D_{32}, and D_{33}*_spec*, which is the same as *a_spec* in Table 3, are independently verified first and then composed into D_2*_real*. The verification of D_2*_real* is performed using the bit-state hashing option, since it fails with an out-of-memory error when the exhaustive search option is used.

Table 2
Experiments on controlled composition.

Name	Type	Depth	States	Transitions	Memory	Time	H/F
e	U	2715	101,694	135,115	95.70	0.35	
d	U	34	167	320	4.50	0.00	
b (*d* + *e*)	C	25,531	2,575,743	4,323,539	107.64	20.70	
g	U	43	111	215	2.19	0.00	
h	U	25,760	904,690	1,611,177	58.04	4.36	
f (*g* + *h*)	C	32,289	1,330,836	2,409,953	99.82	6.41	
c	U	4399	2,699,048	3,700,386	78.30	16.30	
a (b + c+f)	C	OM	OM	OM	OM	OM	
a (*b* + *c* + *f*)	C	52,616	2.14e + 09	3.18e + 09	515.80	5890.00	2.08

Table 3
Experiments on controlled composition and abstraction.

Name	Type	Depth	States	Transitions	Memory	Time	H/F
e	U	2715	101,694	135,115	95.70	0.35	
d	U	34	167	320	4.50	0.00	
*b*_real	C	25,531	2,575,743	4,323,539	107.64	20.70	
*b*_spec	C	763	12,227	19,379	3.58	0.02	
g	U	43	111	215	2.19	0.00	
h	U	25,760	904,690	1,611,177	58.04	4.36	
*f*_real	C	32,289	1,330,836	2,409,953	99.82	6.41	
*f*_spec	C	1331	30,163	68,204	33.58	0.15	
c	U	4399	2,699,048	3,700,386	78.30	16.30	
*c*_f_real	C	88,204	21,246,200	48,395,531	546.63	54.90	79.10
*c*_f_spec	C	19,509	372,043	1,971,796	124.63	7.53	
*a*_real	C	28,414	74,525,803	55,306,400	546.63	69.00	103.34
*a*_spec	C	689	7,964,500	12,382,052	248.75	46.90	

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

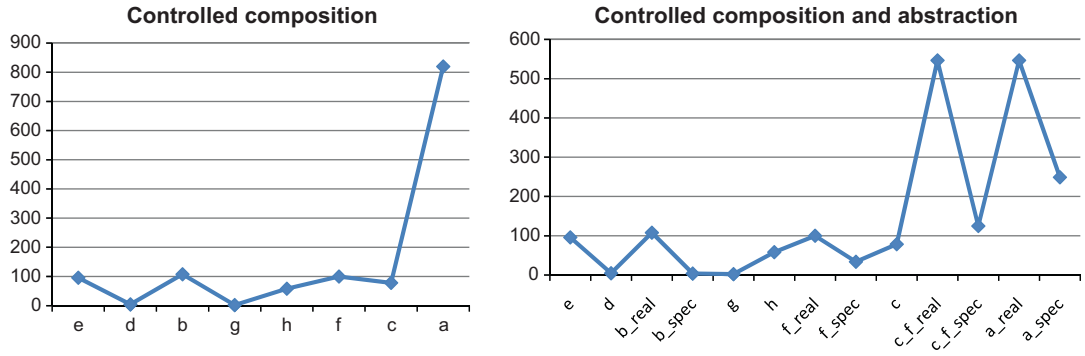


Fig. 16. Scalability of memory consumption.

Table 4
Experiments on high-level abstract components.

Name	Type	Depth	States	Transitions	Memory	Time	H/F
D_31	U	13,826	10,760,942	20,504,583	528.12	96.90	
D_32	U	670	114,512	131,953	16.86	0.20	
D_33_spec	C	689	7,964,500	12,382,052	248.75	46.90	
D_2_real	C	11,692	354,676,100	997,864,410	515.63	2220.00	6.05
D_2_spec	C	9833	12,903,757	28,985,781	397.65	77.30	
C_1	U	254	5290	8718	35.35	0.04	
C_real	C	2240	226,844	250,943	44.23	1.12	
C_spec	C	425	23,155	27,464	3.09	0.08	
B_spec	C	66	95	111	2.50	0.00	
A_real	C	170,886	43,300,574	103,316,890	519.36	131.00	49.59
A_spec	C	159,947	10,544,618	24,503,674	497.41	122.00	

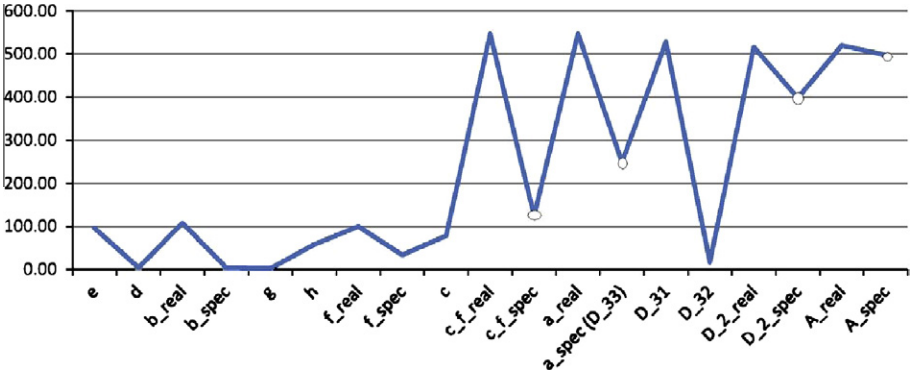


Fig. 17. Memory usage for model checking abstract components of the RadioCountToLeds application.

After applying synchronous abstraction and projection, however, the abstract model D_2_spec of D_2_real is verified using the exhaustive search option with about 400 Mbytes of memory. *A_real* is the realization model of the top-level application component, which is composed of the abstract components *B*, *C*, *D*, *E*, *F*, and *G* using parallel composition. It also results in an out-of-memory error with the exhaustive search option, but its controlled and abstracted version *A_spec* was verified within 500 Mbytes of memory and 122 s.

Fig. 17 illustrates the memory usage graph when successively composing and verifying abstract components bottom-up; the leftmost side shows the bottom-level components, and successive composite components are shown going towards the right side. It is clear that each composition results in a local peak in memory usage (e.g., *c_f_real*, *a_real*,D_2_real, and *A_real*), but this is alleviated after applying controlled composition and abstractions (e.g., *c_f_spec*, *a_spec*, D_2_spec, and *A_spec*). We note that the growth of memory usage is linear when we consider only the specification models.

8. Related work

Formal methods are frequently used for quality assurance in the development of component-based systems [36,46,25]. This section summarizes existing approaches related to this work in four categories; (1) interface theories for defining the semantics of compositions, (2) approaches for improving the scalability of model checking, (3) approaches for modeling and adaptation, and (4) domain-specific verification approaches for TinyOS. We also discuss some of the existing program verification tools and their possible roles in our approach.

8.1. Interface theories

There have been numerous studies on interface theories for model checking components [34,35,28,22,7,21]. [35] proposed an operational model for the coordination language Reo for specifying

component connectors. It provides a composition operator that joins two connectors, which corresponds to parallel composition, and defines port hiding, which is similar to projection abstraction in this work. Their work is focused on defining semantics for the Reo connectors, which are classified into synchronous channels for simultaneous sending and receiving of messages and FIFO channels for asynchronous connection. Contrary to this, our work deals with synchronous message passing at the application level, depending on whether an operational call requires a response or not from the service provider, regardless of the type of connectors used.

Ref. [28] proposed a theoretical framework for modeling component-based systems by formally defining the notions of components, connectors, and the important properties of interaction systems. They emphasize the necessity of concepts and tools for integrating synchronous and asynchronous components as well as different communication mechanisms, such as communication via shared variables, signals, or rendezvous. Among the three models composing their framework for component-based modeling – a behavioral model, an interaction model, and an execution model – the execution model deals with synchronous and asynchronous executions, where high-level abstract layers adopt asynchronous interaction, while strong synchronous execution with priority is assumed in the lower-level abstract layers. In comparison, our approach treats synchronous and asynchronous execution semantics at the same level.

Giese et al. [22] used the notion of connectors to define domain-specific patterns for real-time UML designs. Their approach is based on the synchronous communication assumption that sending and receiving happens within the same time step. We eliminate this assumption in our approach, allowing messages to be buffered and processed in different time steps.

8.2. Scalability improvement

There are two major approaches to improving scalability in model checking components: one is compositional minimization [11,13,29,51], and the other is compositional model checking [9,30,42,18].

Compositional minimization tries to minimize the composition itself before applying model checking; given a parallel composition of two components $C_1 \| C_2$, these approaches reduce each component C_i either by using property-based model extraction [13], predicate-based input-restricting minimization [51], or a bisimulation equivalence relation [11,29] with respect to the composition interface. Our work is closest to the last approach, but we added transition reduction using the information about the communication mechanism and systemized the reduction process.

In compositional model checking, the verification problem p, $A|B \models q$ of a composition of two components A and B is divided into two problems of verifying p, $A \models r$ and r, $B \models q$. Assuming that the necessary and sufficient premise r satisfies both problems, the compositional verification problem can be reduced to the verification problem for one component. Nevertheless, finding such premises is known to be difficult [18] and most recent related research activities have focused on automatically finding good assumptions [9,30,42].

Automatic generation of verification environments for model checking components has also been an important issue for efficient model checking [19,50], but the focus is on finding the smallest environment of a component based on its interface specification without considering a specific composition case.

8.3. Modeling and adaptation

It is worth noting a couple of existing model-based adaptation techniques for assembling mismatching components. Nejati et al. [43] presented heuristics and/or a systematic algorithm for matching and merging statecharts. Canal et al. [12] proposed synchronous products and reordering of messages to resolve mismatched communication behavior between two components. Ziadi et al. [52] defines an algebraic framework for composing statecharts and then shows how to synthesize a statechart from UML 2.0 sequence diagrams. These can be adopted in our framework for more sophisticated composition, but our current work does not consider mismatching component behaviors; mismatched behaviors are identified in the verification process as counterexamples of communication consistency.

Pelliccione et al. [45] proposed an automatic and verifiable component assembly method using CHARMY [44]. Their approach first verifies and refines the software architecture with respect to a set of properties using the CHARMY tool, and then locally assembles the actual components into the verified architecture by automatically generating adaptor code that addresses behavioral mismatches. The CHARMY approach deals with a framework for model-driven component development and verification that has a similar goal as the MARMOT framework [14]. Their approach formally defines relationships among components, state diagrams, and sequence diagrams. However, it does not deal with refinements and abstractions. The scalability issue of model checking, the main concern of our work, is not treated either.

Baresi et al. [4] proposed a domain-specific model checker equipped with an underlying communication infrastructure that has been overlooked by general-purpose model checkers. This includes mechanisms for message ordering, filtering, subscription delay, replies, queue size, and queue drop policy. In comparison, our control model is concerned with the application-level message passing mechanism; we adopt the communication constructs provided by the model checker SPIN for the underlying communication infrastructure.

8.4. Model checking TinyOS

A couple of modeling and analysis techniques specific to TinyOS are available, but none of them mentions performance issues. Archer et al. [3] proposed interface contracts for TinyOS to reduce programming errors caused by misunderstanding interfaces, and Basu et al. [5] presented a model construction methodology applied to TinyOS-based networks, which focuses on how to construct component models for TinyOS modules. It is a good reference for constructing an abstract component model from the code, but higher-level abstraction is not considered. Völgyesi et al. [48] suggested a verification approach based on interface automata for component compositions, especially targeting TinyOS components; it proposed an interface language that specifies the external behavior of a component, and checks interface compatibility using interface automata and their composition rules. As the authors noted in the paper, most of the TinyOS components do not interact asynchronously, and, thus, their approach to modeling TinyOS components in PROMELA was not efficient in verification. Our approach addresses exactly this issue by introducing a control model for synchronous message passing.

8.5. Tools for program verification

Model-checking software programs have been supported by numerous tools [16,47,20,6]. Most notably, CBMC [16] takes ANSI C/C++ programs, converts them into formal models in a conjunctive normal form, and verifies basic properties, such as pointer misuse and array bound checking, as well as user-defined properties specified as assertions, using SAT-solver. Java PathFinder [47] is a model checker for programs written in Java, which is based on an explicit model-checking technique. VCC [20] supports modular

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001

verification of C programs using SMT-solver as its underlying verifier.

These are quite useful tools that can be applied in various application domains [10,24,23] if the purpose is code verification. Nevertheless, program verifiers do not play a major role when the goal is bottom-up verification and composition for reverse-engineering abstract components where behavioral model extraction is required. Moreover, existing code verifiers cannot handle embedded software directly due to several reasons: (1) they cannot directly handle hardware-dependent code written in inline assembly code, (2) they need environment models to construct a closed verification model, which requires manual modeling [39], and (3) each code verifier has its own limitation. For example, infinite loops cannot be handled without manual intervention in CBMC, analysis of arrays is not precise in BLAST [38], and VCC requires manual annotations on pre- and post-conditions for each function to be verified, which is not a trivial task. Therefore, abstraction and model extraction is a necessary process before applying code verifiers. It is possible to use those tools as the back-end verifier in our approach instead of Spin, but a model extraction process is required anyway and the extracted model has to be expressed in a specific implementation language such as C. We chose the Spin modeling language since it makes more sense to make the model language-independent.

9. Discussion

This work introduced a method for the systematic construction of verification models using controlled composition and abstraction. A control model for the inter-component message passing mechanism is proposed together with abstraction methods for systematically extracting a higher-level abstract component from the composition of lower-level components. Two abstraction techniques, synchronized abstraction and project abstraction, are adopted from process algebra [32,41], but the novelty of the technique lies in the systematic application of the operations in the bottom-up construction of abstract components with respect to port bindings and port dependencies.

As demonstrated with the experiments, the proposed approach is effective in reducing verification cost while retaining the comprehensiveness of the model checking technique. We note that abstraction techniques are applied to a composition only after the successful verification of the composite components, which reduces the risk of overlooking the important behavior of a component, especially its communication behavior.

Though improving model checking scalability is the most important theme of this work, one of the major motivations is to extract reusable and verified components from code that can be reused in model-driven development processes. Verified underlying components, such as operating system components, with their abstract representations, can be reused in the top-down modeling and verification process for any applications on top of the operating system. Therefore, our approach emphasizes the successive model extraction process as well as the verification aspect.

The main contribution of the suggested approach is twofold: First, a systematic compositional minimization method specifically designed for single processor systems is introduced. Second, an iterative behavioral model extraction technique is provided. The approach is demonstrated on a TinyOS application showing the reduction of memory consumption in model checking; a linear growth of memory usage is observed instead of the typical exponential growth. However, it also shows that controlled composition alone does not make model checking scale up to the verification of the top-level abstract component; as shown in the experiment result, a couple of realization compositions required memory and time beyond our experimental setting, which resulted in using a non-exhaustive search option. Though overall performance is promising, further investigation for improving the verification performance, especially for the realization composition, is necessary. We also showed that our approach guarantees the exactness of the verification result for state properties, but false negatives are possible for path properties. Therefore, the suggested abstraction techniques are not suitable for path properties such as hard realtime issues.

The approach is generally applicable to the verification of functional correctness and interaction consistency for each composition. Once this bottom-up composition and verification is done, the constructed composition tree with behavioral specifications can be used for checking system-level properties. This may require top-down decomposition of global properties and/or a systematic identification of physical components related to the system-level errors. We leave these issues to future work.

Acknowledgments

This work was supported by National Research Foundation of Korea Grant funded by the Korean Government (2010-0017156) and the Engineering Research Center of Excellence Program of the Korean Ministry of Education, Science and Technology(MEST)/ National Research Foundation(NRF) (Grant 2011-0000978).

References

[1] TinyOS website. <http://www.tinyos.net/>.
[2] S. Anand, C.S. Pasareanu, W. Visser, Symbolic execution with abstraction, Software Tools for Technology Transfer 11 (1) (2008) 53–67.
[3] Will Archer, Philip Levis, John Regehr, Interface contracts for tinyOS, in: Information Processing in Sensor Networks, April 2007, pp. 158–165.
[4] L. Baresi, C. Ghezzi, L. Mottola, Loupe: verifying publish–subscribe architecture with a magnifying lens, IEEE Transactions on Software Engineering 37 (2) (2010) 228–246.
[5] A. Basu, L. Mounnier, M. Poulhies, J. Pulou, J. Sifakis, Using BIP for modeling and verification of networked systems – a case study on tinyOS-based networks, in: 6th IEEE International Symposium on Network Computing and Application, July 2007, pp. 257–260.
[6] Dirk Beyer, Thomas A. Hensinger, Ranjit Jhala, Rupak Majumdar, The software model checker blast: applications to software engineering, International Journal on Software Tools for Technology Transfer 9 (5) (2007).
[7] Simon Bliudze, Joseph Sifakis, The algebra of connectors – structuring interaction in BIP, IEEE Transactions on Computers (2008).
[8] Mihaela Gheorghiu Bobaru, Dimitra Giannakopoulou, Corina S. Pasareanu, Refining interface alphabets for compositional verification, in: 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2007, pp. 292–307.
[9] Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement, in: 20th International Conference on Computer Aided Verification, 2008, pp. 135–148.
[10] Doina Bucur, Marta Z. Kwiatowska. Poster abstract: software verification for TinyOS, in: 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, 2010.
[11] D. Bustan, O. Grumberg. Modular minimization of deterministic finite-state machines, in: Proceedings of the 6th International Workshop on Formal Methods in Industrial Critical Systems, 2001, pp. 163–178.
[12] Carlos Canal, Pascal Poizat, Gwen Salauen, Model-based adaptation of behavioral mismatching components, IEEE Transactions on Software Engineering 34 (4) (2008) 546–563.
[13] M. Chiodo, T.R. Shiple, A.L. Sangiovanni-Vincentelli, R.K. Brayton, Automatic compositional minimization in CTL model checking, in: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 1992.
[14] Yunja Choi, Christian Bunse, Design verification in model-based μ-controller development using an abstract component, Software and Systems Modeling 10 (1) (2011) 91–115.
[15] Edmund Clarke, Armin Biere, Richard Raim, Yunshan Zhu, Bounded model checking using satisfiability solving, Formal Methods in System Design 19 (1) (2001).
[16] Edmund Clarke, Daniel Kroening, Flavio Lerda, A tool for checking ANSI-C programs, in: 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2004.
[17] Edmund M. Clarke, Orna Grumberg, Doron Peled, Model Checking, MIT Press, 1999.

[18] Jamieson M. Cobleigh, George S. Avrunin, Lori A. Clarke, Breaking up is hard to do: an evaluation of automated assume-guarantee reasoning, ACM Transactions on Software Engineering and Methodology (2008).
[19] C. Colby, P. Godefroid, L.J. Jagadeesan, Automatically closing open reactive programs, ACM SIGPLAN Notices (1998).
[20] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, W. Shulte, VCC: contract-based modular verification of concurrent C, in: 31st International Conference on Software Engineering, 2008.
[21] Luca de Alfaro, Thomas A. Henzinger, Interface theories for component-based design, in: Proceedings of the First International Workshop on Embedded Software, 2001.
[22] Holger Giese, et al., Towards the composition verification of real-time UML designs. In Proceedings of the 9th European Software Engineering Conference/ 11th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2003.
[23] J. Penix et al., Verifying time partitioning in the DEOS scheduling kernel, Formal Methods in Systems Design Journal 26 (2) (2005).
[24] Lucas Cordeiro, et al., Semiformal verification of embedded software in medical devices considering stringent hardware constraints, in: International Conference on Embedded Software and Systems, 2009.
[25] Luis Gomes, et al., Towards usage of formal methods within embedded systems co-design, in: 10th IEEE International Conference on Emerging Technologies and Factory Automation, September 2005.
[26] Gerald C. Gannod, Betty H.C.Cheng, A suite of tools for facilitating reverse engineering using formal methods, in: 9th International Workshop on Programming Comprehension, 2001, pp. 221–232.
[27] D. Gay, P. Levis, R. Behren, et al., The nesC language: a holistic approach to networked embedded systems, in: Conference on Programming Language Design and Implementation, June 2003, pp. 1–11.
[28] Gregor Goessler, Sussane Graf, Mila Majster-Cederbaum, M. Martens, Joseph Sifakis, An approach to modelling and verification of component based systems, in: SOFSEM 2007, LNCS, vol. 4362, 2007, pp. 295–308.
[29] S. Graf, B. Steffen, G. Luttgen, Compositional minimization of finite state systems using interface specifications, Formal Aspects of Computing 8 (1996) 607–616.
[30] A. Gupta, K.L. McMillan, Z. Fu, Automated assumption generation for compositional verification, Formal Methods in System Design 32 (2008) 285–301.
[31] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274.
[32] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[33] Gerard J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley Publishing Company, 2003.
[34] Graham Hughes, Tevfik Bultan, Interface grammars for modular software model checking, IEEE Transactions on Software Engineering 34 (5) (2008) 614–632.
[35] Mohammad Izadi, Marcello M. Bonsangue, Dave Clarke, Modeling component connectors: Synchronisation and context-dependency, in: 6th IEEE International Conference on Software Engineering and Formal Methods, 2008.
[36] Steven D. Johnson, Formal methods in embedded design, IEEE Computer 36 (11) (2003) 104–106.
[37] M.U. Khan, K. Geihs, et al., Model-driven development of real-time systems with UML 2.0 and C, in: Proceedings of the 3rd International Workshop on Model-based Methodologies for Pervasive and Embedded Software at the 13th IEEE International Conference on Engineering, 2006.
[38] M. Kim, Y. Kim, H. Kim, A comparative study of software model checkers as unit testing tools: an industrial case study, IEEE Transactions on Software Engineering 37 (2) (2011).
[39] Moonzoo Kim, Yunja Choi, Yunho Kim, Hotae Kim. Formal verification of a flash memory device driver – an experience report, in: 15th International SPIN Workshop on Model Checking Software, 2008.
[40] Philip Levis, David Gay, TinyOS Programming, Cambridge University Press, 2009.
[41] Robin Milner, Communicating and Mobile Systems: The π-calculus, Cambridge University Press, 1999.
[42] Wonhong Nam, P. Madhusudan, Rajeev Alur, Automatic symbolic compositional verification by learning assumptions, Formal Methods in System Design 32 (3) (2008) 207–234.
[43] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, Parmela Zave, Matching and merging of statecharts specifications, in: 29th International Conference on Software Engineering, 2007.
[44] P. Pelliccione, Paola Inveradi, Henry Muccini, CHARMY: a framework for designing and verifying architectural specifications, IEEE Transactions on Software Engineering 35 (3) (2009) 325–346.
[45] P. Pelliccione, M. Tivoli, A. Bucchiarone, A. Polini, An architectural approach to the correct and automatic assembly of evolving component-based systems, The Journal of Systems and Software 81 (12) (2008) 2237–2251.
[46] Oscar R. Ribeiro, Joao M. Fernandes, Luis F. Pinto, Model checking embedded systems with PROMELA, in: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005.
[47] W. Visser, K. Havelund, G. Brat, S. Park, Model checking programs, in: 15th IEEE International Conference on Automated Software Engineering, September 2000.
[48] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, Á. Lédeczi, Software composition and verification for sensor networks, Science of Computer Programming 56 (2005) 191–210.
[49] Fei Xie, James C. Browne, Verified systems by composition from verified components, in: Proceedings of Joint Conference ESEC/FSE, 2003.
[50] Haiqiong Yao, Hao Zheng, Automated interface refinement for compositional verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28 (3) (2009) 433–446.
[51] F. Zaraket, J. Baumgartner, A. Aziz, Scalable compositional minimization via static analysis, in: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 2005.
[52] Tewfik Ziadi, Loïc Helouët, Jean-Marc Jezequel. Revisiting statechart synthesis with an algebraic approach, in: 26th International Conference on Software Engineering, 2004.

Please cite this article in press as: Y. Choi, M. Kim, Controlled composition and abstraction for bottom-up integration and verification of abstract components, Inform. Softw. Technol. (2011), doi:10.1016/j.infsof.2011.08.001